# UNIT–V

***Machine Independent Optimization. The principle sources of Optimization peephole Optimization, Introduction to Date flow Analysis, Foundations of Data-Flow Analysis, Constant Propagation, Partial Redundancy Elimination, Loops in Flow Graph.***

**MACHINE INDEPENDENT CODE OPTIMIZATION:**
The optimization which is done on an intermediate code is called machine independent code optimization because intermediate code does not depend on any target machine.

## 5.1. The Principle Sources of Optimization

A compiler optimization must preserve the semantics of the original program. When a programmer implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm. A compiler knows only how to apply relatively low-level semantic transformation.

### 5.1.1. Causes of Redundancy

There are many redundant operations in a typical program. Sometimes the redundancy is available at the source level. In most languages (other than C or C++, where pointer arithmetic is allowed), programmers have no choice but to refer to elements of an array or fields in a structure through accesses like A[i][j] or X→f1.
As a program is compiled, each of these high-level data-structure accesses expands into a number of low-level arithmetic operations, such as the computation of the location of the (i,j)th element of a matrix A. Accesses to the same data structure often share many common low-level operations. Programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves. The programmers only access data elements by their high-level names. If the compiler eliminates the redundancies then the resultant code will be efficient and easy to maintain.

## 5.2 Semantic Preserving Transformation:

There are number of optimization techniques that a compiler can use to improve the performance of the program without changing its output. The optimization technique should not influence the semantics of the program. Common-sub expression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such semantic preserving transformations

### 5.2.1 Common Sub-Expression Elimination:

The code can be improved by eliminating common sub expressions from the code. An expression whose value is previously computed and the values of variables in the expression are not changed, since its computation can be avoided to by using the earlier computed value.
The below example shows the optimized code after eliminating the common sub expressions.

```
a := b + c                         a := b + c
b := a – d      ----------->       b := a - d
c := b + c                         c := b + c
d := a – d                         d := b
```

**Fig (a):** Before Elimination **Fig(b)** After Common sub exp elimination

In above code d=a-d can be replaced by d=b. This technique optimizes the intermediate code. If we eliminate common sub-expression in one block of code then it is called local common sub-expression elimination. If we eliminate common sub-expressions among multiple blocks then it is called Global Common sub-expression elimination. The above example shows local common sub-expression elimination. In the below example this optimization is performed first locally and then globally.

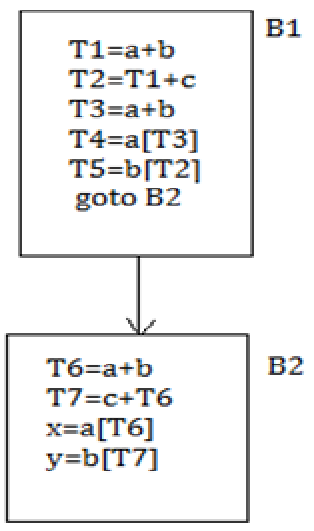


Fig.a) Before Global Common sub expression elimination

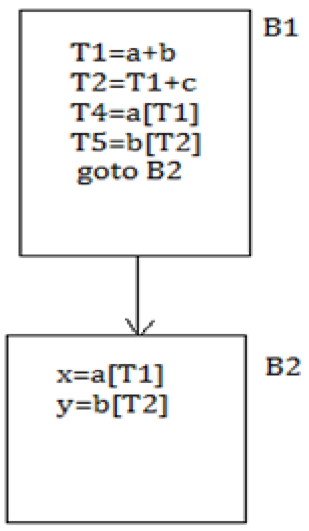


Fig.b) After Elimination

**Example 2: Quick Sort Example**

In this example a fragment of a sorting program called ***quicksort*** *is used* to illustrate several important code-improving transformations.

```
void quicksort(int m, int n)
/ *recursively sorts a[m] through a[n] */
{
        int i , j ;
        int v, x;
        if (n <= m) return;
        / * fragment begins here */
        i = m - 1 ; j = n ; v = a[n] ;
        while (1) {
                do i = i + 1 ; while ( a[i] < v );
                do j = j - 1 ; while ( a[j] > v );
                i f (i>= j) break;
                x = a[i]; a[i] = a[j]; a[j] = x; /* swap a [i],a [j] */
                }
        x = a [i]; a[i] = a[n]; a[n] = x; /* swap a[i],a[n] */
        /* fragment ends here */
        quicksort(m,j) ; quicksort(i+1,n );
}
```

***Figure 5.1: C code for quicksort***

Intermediate code for the marked fragment of the program is shown in Fig. 5.2.

```
(1)      i = m-1                    (16)     t7 = 4*i
(2)      j = n                      (17)     t8 = 4*j
(3)      t1 = 4*n                   (18)     t9 = a[t8]
(4)      v = a[t1]                  (19)     a[17] = t9
(5)      i = i+1                    (20)     t10 = 4*j
(6)      t2 = 4*i                   (21)     a[t10] = x
(7)      t3 = a[t2]                 (22)     goto (5)
(8)      if t3<v goto (5)           (23)     t11 = 48i
(9)      j = j-1                    (24)     x = a[t11]
(10)      t4 = 4*i                  (25)     t12 = 4*i
(11)      t5 = a[t4]                (26)     t13 = 4*n
(12)      if t5>v goto (9)          (27)     t14 = a[t13]
(13)      if i>=j  goto (23)        (28)     a[t12] = t14
(14)      t6 = 4*i                  (29)     t15 = 4*n
(15)      x=a[t6]                   (30)     a[t15] = x
```

***Figure 5.2: Three-address code for fragment in Fig. 6.1***

In this example we assume that integers occupy four bytes. The assignment x = a[i] is translated into the two three-address statements

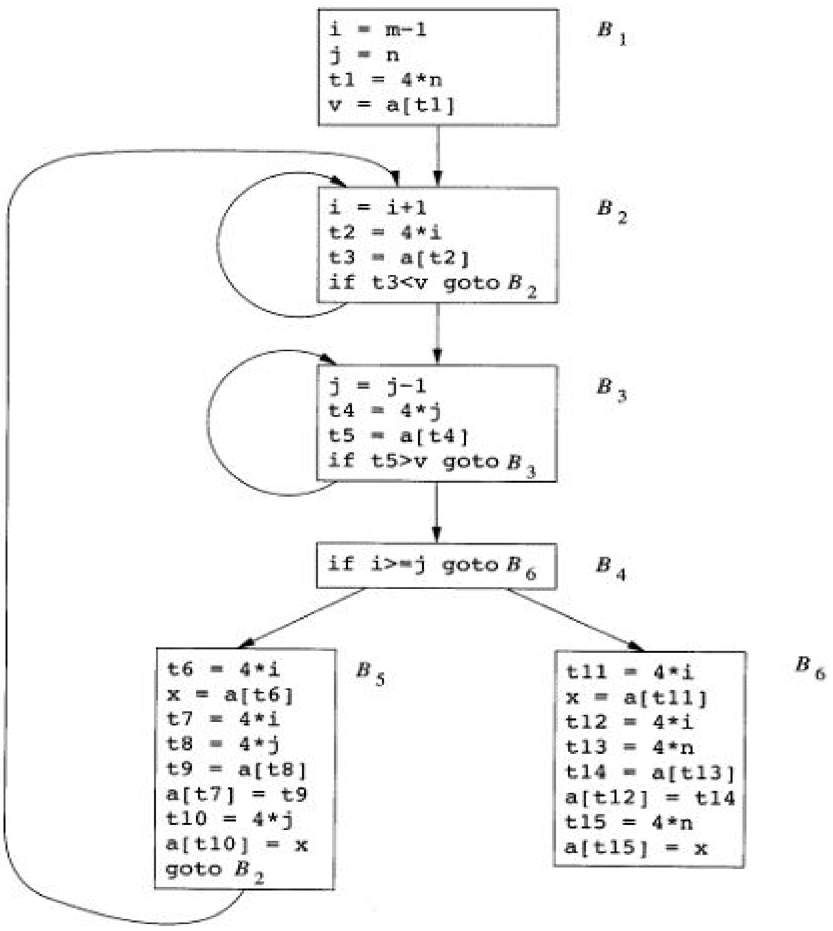
```
t6 = 4*i
x = a[t6]
```

as shown in steps (14) and (15).
Similarly, a[ j ] = x becomes

```
t10 = 4*j
a[t10] = x
```

in steps (20) and (21).

Figure 5.3 is the flow graph for the program in Fig. 5.2. Block *B1* is the entry node. All conditional and unconditional jumps to statements in Fig. 5.2 have been replaced in Fig. 5.3 by jumps to the block of which the statements are leaders. In Fig. 5.3, there are three loops. Blocks *B2* and B3are loops by themselves. Blocks *B2, B3, B4,* and *B5* together form a loop, with *B2* the only entry point.



```
B1:
i = m-1
j = n
t1 = 4*n
v = a[t1]

B2:
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2

B3:
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3

B4:
if i>=j goto B6

B5:
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2

B6:
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

***Figure 5.3: Flow graph for the quicksort fragment***

```
t6 = 4*i          t6 = 4*i
x = a[t6]         x = a[t6]
t7 = 4*i          t8 = 4*j
t8 = 4*j          t9 = a[t8]
t9 = a[t8]        a[t6] = t9
a[t7] = t9        a[t8] = x
t10 = 4*j         gotoB2
a[t10] = x
goto B2
```

***(a) Before.*** ***(b) After.***

***Figure 5.4: Local common-subexpression elimination***

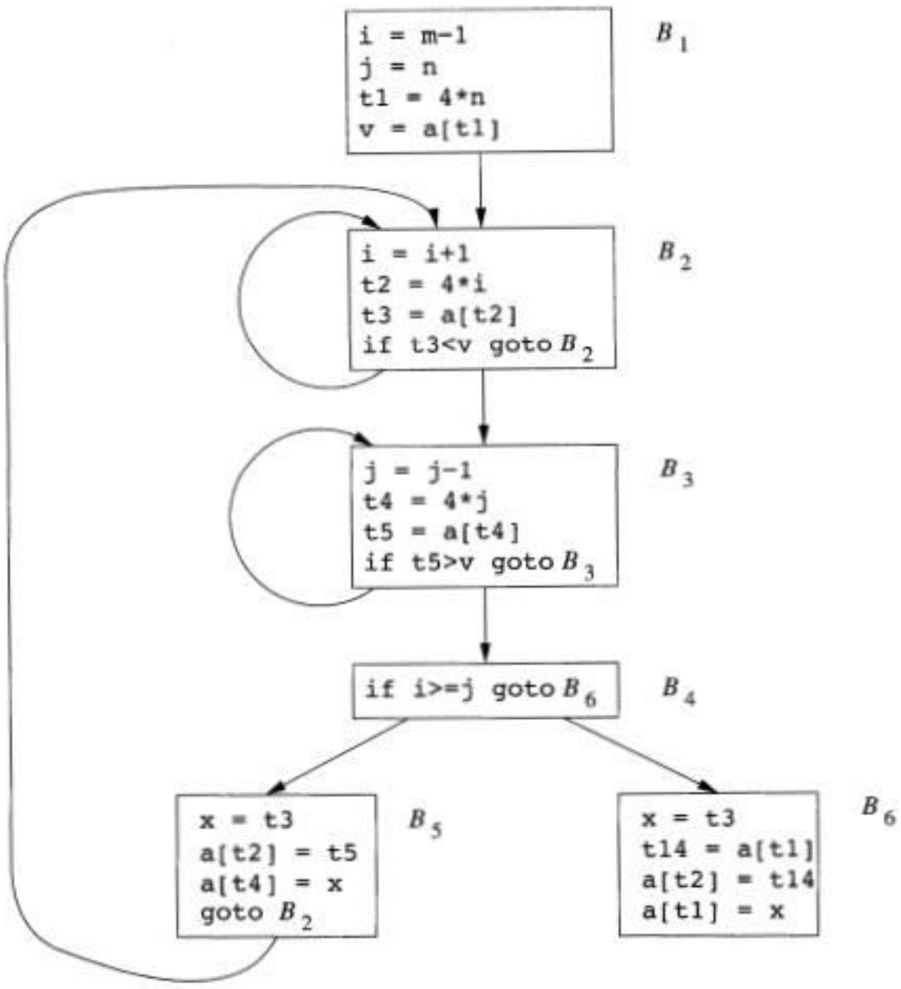


***Figure 5.5: B5 and B6 after common-subexpression elimination***

### 5.2.2 Constant Folding:

This is an optimization technique that evaluates constant expressions at compile time and replaces such expressions by their computed values.
In this technique expressions with constant operands can be evaluated at compile time, thus improving the run-time performance and reducing the code size by avoiding evaluation at compile-time.

**Example:**

In the code fragment below, the expression (3 + 5) can be evaluated at compile time and replaced with the constant 8.

```
int f (void)
{
return 3 + 5;
}
```

Below is the code fragment after constant folding.

```
int f (void)
{
return 8;
}
```

**Notes:**

Constant folding is a relatively easy optimization.
Programmers generally do not write expressions such as (3 + 5) directly, but these expressions are relatively common after macro expansion and other optimizations such as constant propagation.

### 5.2.3 Copy Propagation:

It is the process of replacing the occurrences of targets of direct assignments with their values. A direct assignment is an instruction of the form x = y, which simply assigns the value of y to x.

From the following code:

```
y = x
z = 3 + y
```

Copy propagation would yield:

```
z = 3 + x
```

**Example 2: Quick Sort Example**

The idea behind the copy-propagation transformation is to use *v* for u, wherever possible after the copy statement u = v. For example, the assignmentx = t3 in block *B5* of Fig. 5.5 is a copy. Copy propagation applied to *B5* yields the code in Fig. 5.7. This change may not appear to be an improvement, but, it gives us the opportunity to eliminate the assignment to *x.*

```
x = t3
a[t2] = t5
a[t4] = t3
gotoB2
```

***Figure 5.7: Basic block B5 after copy propagation***

## 5.2.4 Dead Code Elimination:

**Dead code elimination** (also known as **dead code removal**, **dead code stripping**, or **dead code strip**) is an optimization technique to remove the code which does not affect the program results. Removing such code has two benefits: it shrinks the program size and it allows the running program to avoid executing irrelevant operations, which reduces its running time. *Dead code* includes code that can never be executed (*unreachable code*), and code that only affects *dead variables*, that is, variables that are irrelevant to the program.

Consider the following example written in C.

```
int foo(void)
{
int a =24;
int b =25;/* Assignment to dead variable */
int c;
  c = a <<2;
return  c;
  b =24; /* Unreachable code */
return 0;  }
```

Simple analysis of the uses of values would show that the value of b after the first assignment is not used inside foo. Furthermore, b is declared as a local variable inside foo, so its value cannot be used outside foo. Thus, the variable b is *dead* and an optimizer can reclaim its storage space and eliminate its initialization.

Furthermore, because the first return statement is executed unconditionally, no feasible execution path reaches the second assignment to b. Thus, the assignment is *unreachable* and can be removed.

**Example 2:**

In the example below, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to global is dead, and the third assignment to global is unreachable; both can be eliminated.

```
int global;
void f ()
{
int i;
i = 1;         /* dead store */
global = 1;    /* dead store */
global = 2;
return;
global = 3;     /* unreachable */
}
```

Below is the code fragment after dead code elimination.

```
int global;
void f ()
{
global = 2;
return;
}
```

**Example 3: Quick Sort Example**

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to *x* and transforms the code in Fig 5.7 into

*a[t2] = t5*
*a[t4] = t3*
*gotoB2*

This code is a further improvement of block *B5* in Fig. 5.5.

### 5.2.5. Strength Reduction:

**Strength reduction** is an optimization technique in which expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing. By doing this the execution speed can be increased.

Example 1: Consider the following code

```
for(i=1;i<=5;i++)
{
x=4*i;
}
```

The instruction ***x=4*i*** in the loop can be replaced by equivalent additions instruction as ***x=x+4*;**

Code after strength reduction is shown below.

```
x=0;
for(i=1;i<=5;i++)
x=x+4;
```

**Example 2: Quick Sort**

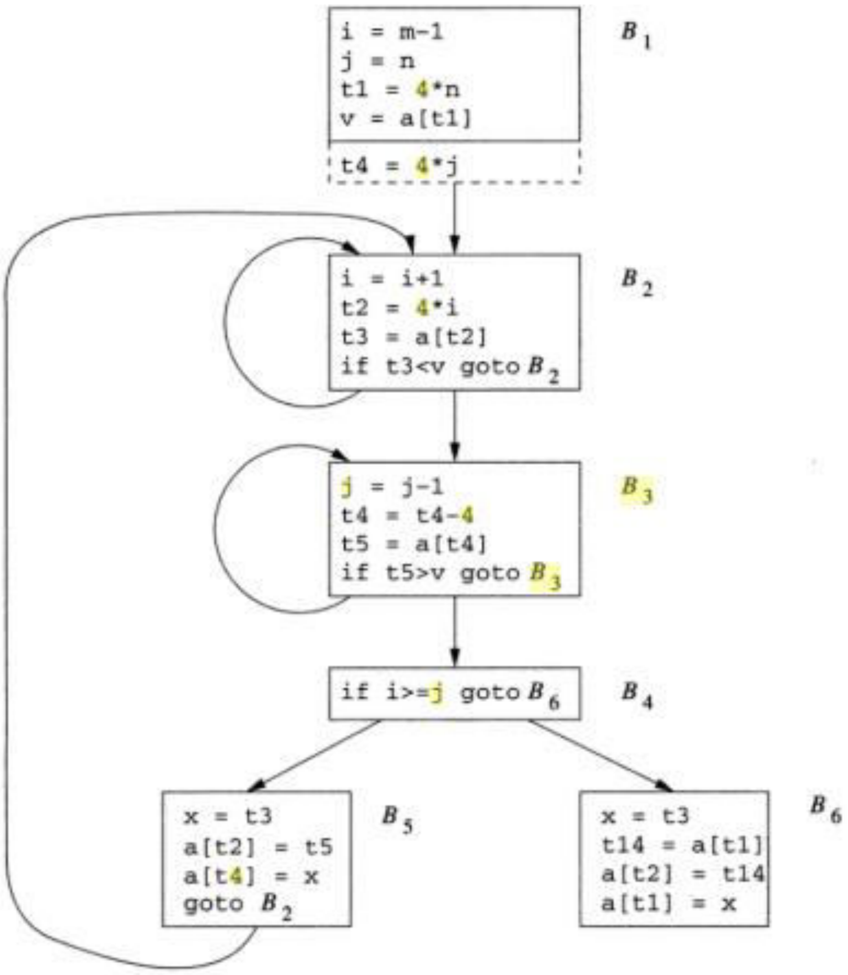


*Figure 5.8: Strength reduction applied to 4 * j in block B3*

## 5.3. Loop Optimization:

Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

There are three techniques:

1. Code Motion
2. Elimination of induction variables
3. Strength reduction

**1. Code Motion:**

Code motion reduces the number of instructions in a loop by moving instructions outside a loop. It moves loop invariant computations i.e, those instructions or expressions that result in the same value independent of the number of times a loop is executed and places them at the beginning of the loop. The relocated expressions become an entry for the loop.

**Example:**

```
While(X!=n-2)
{
X=X+2;
}
```

here the expression n-2 is a loop invariant computation i.e, the value evaluated by this expression is independent of the number of times the while loop is executed. In other words the value of n remains unchanged. The code relocation places the expressions n-2 before the while loop begins as shown below.

```
M=n-2;
While(X!=M)
{
X=X+2;
}
```

2. **Elimination of Induction Variables:**
   An induction variable is a loop control variable or any other variable that depends on the induction variable in some fixed way. It can also be defined as variable which is incremented or decremented by a fixed number in a loop each time the loop is executed. If there are two or more induction variables in a loop then by the induction variable elimination process all can be eliminated except one.
   **Example:**
   ```
   void fun(void)
   {
   inti,j,k;
   for(i=0,j=0,k=0;i<10;i++)
   a[j++]=b[k++];
   return;
   }
   ```
   In the above code there are three induction variables i,j and k which take on the values 1,2,3…10 each time through the beginning of the loop. Suppose that the values of the variables j and k are not used after the end of the loop then we can eliminate them from the function fun() by replacing them by variable i.
   After induction variable elimination, the above code becomes
   ```
   void fun(void)
   {
   inti,j,k;
   for(i=0;i<10;i++)
   a[i]=b[i];
   return;
   }
   ```
   Thus induction variable elimination reduces the code and improves the run time performance.
3. **Strength Reduction:**
   **Strength reduction** is an optimization technique in which expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing. By doing this the execution speed can be increased. For example consider the code
   ```
   for(i=1;i<=5;i++)
   {
   x=4*i;
   }
   ```
   The instruction **x=4*i** in the loop can be replaced by equivalent additions instruction as ***x=x+4***;

## 5.4. Instruction Scheduling:

In this technique the instructions are rearranged to generate an efficient code using minimum number of registers. By changing the order in which computations are done we can obtain the object code with minimum cost. The technique of code optimization done by rearrangement of some sequence of instructions is called Instruction Scheduling.

**Example:**

```
TAC
T1= a+b
T2=c-d
T3=e+T2
T4=T1+T3
```

```
TARGET CODE
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, T1
MOV e, R0
ADD R0, R1
MOV T1, R0
ADD R1, R0
MOV R0, T4
```

**Fig (a): Before Instruction Scheduling.**

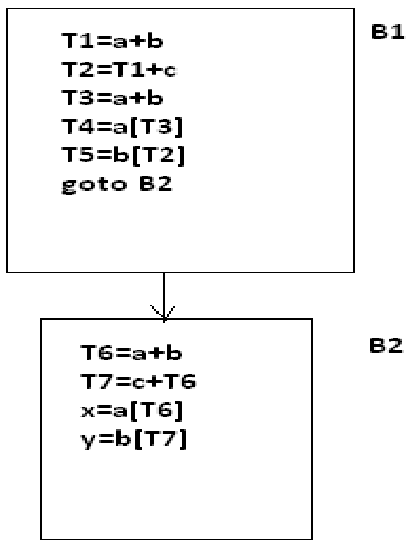
```
TAC
T2=c-d
T3=e+T2
T1= a+b
T4=T1+T3
```

```
TARGET CODE
MOV c, R0
SUB d, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD b, R0
ADD R1, R0
MOV R0, T4
```

**Fig (b): After Instruction Scheduling**

In above diagram TAC with its equivalent Target code is shown.
Now if we rearrange the instructions of TAC then the Target code will get reduced. In the first case the assembly code contains 10 lines. After rearranging the TAC then the assembly code contains 8 Lines. So by rearranging some sequence of instructions we can generate an efficient code using minimum number of registers.

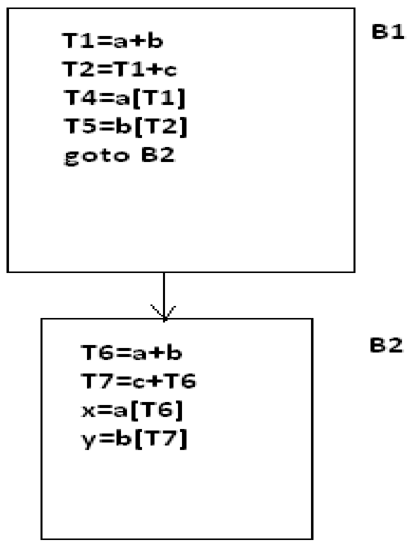
**5.5. Interprocedural Optimization:**
It is a kind of code optimization in which collection of optimization techniques are used to improve the performance of the program. IPO reduces or eliminates duplicate calculations, inefficient use of memory and simplify the loops. IPO may reorder instructions for better memory utilization. IPO also checks the branches or code that never get executed and removes them from the program (dead code elimination).
In this technique first we apply optimization techniques on each block i.e local optimization and then we apply optimization techniques on all the blocks of program i.e global optimization.
The example for this techniques is shown in the figure below.



```
B1
T1=a+b
T2=T1+c
T3=a+b
T4=a[T3]
T5=b[T2]
goto B2

B2
T6=a+b
T7=c+T6
x=a[T6]
y=b[T7]
```

Fig a: Before local Optimization



```
B1
T1=a+b
T2=T1+c
T4=a[T1]
T5=b[T2]
goto B2

B2
T6=a+b
T7=c+T6
x=a[T6]
y=b[T7]
```

fig b After local Optimization

First we apply local optimization on B1 and B2 independently. As shown in above figure. B1 contains common sub-expression and copy propagations they are eliminated. After B1, optimization is performed independently on B2.

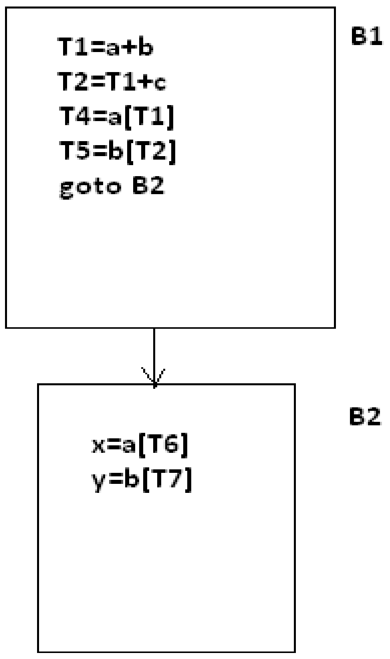


Fig. After Global Optimization

After performing local optimization, global optimization is performed on B1 and B2. In this case B2 contains T6=a+b and T7=c+T6 whose values are already calculated in B1. So instead of recomputing these expressions we can use T1 and T2 directly in B2. After global Optimization the modified code is shown above figure.

## 5.6. Peephole Optimization

A simple but effective technique for locally improving the target code is ***peephole optimization,*** which is done by examining a sliding window of target instructions (called the *peephole)* and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

The following peephole optimization techniques may be applied to improve the performance of the target program:

- Redundant-instruction elimination
- Elimination of Unreachable Code.
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

### i) Eliminating Redundant Loads and Stores

If we see the instruction sequence

*LD a, R0*
*ST R0, a*

in a target program, we can delete the store instruction. Note that if the store instruction has a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. The two instructions have to be in the same basic block for this transformation to be safe.

### ii) Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed.

***Goto L2***
***Print Debug Information***
***L2:***

*In above example Print Statement can be eliminated.*

**iii) Flow-of-Control Optimizations**

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

**goto L1**
**...**
**Ll: goto L2**

by the sequence

**goto L2**
**...**
**Ll: goto L2**

If there are now no jumps to L1, then it may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump.

Similarly, the sequence

**If a< b goto L1**
**------**
**L1: goto L2**

can be replaced by the sequence

**If a< b goto L2**
**------**
**L1: goto L2**

**iv) Algebraic Simplification and Reduction in Strength**

The algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as X = X + 0 and X = X * 1 in the peephole. Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, $x^2$ can be implemented by using x*x. Similarly x*x can be implemented by using x+x.

**v) Use of Machine Idioms**

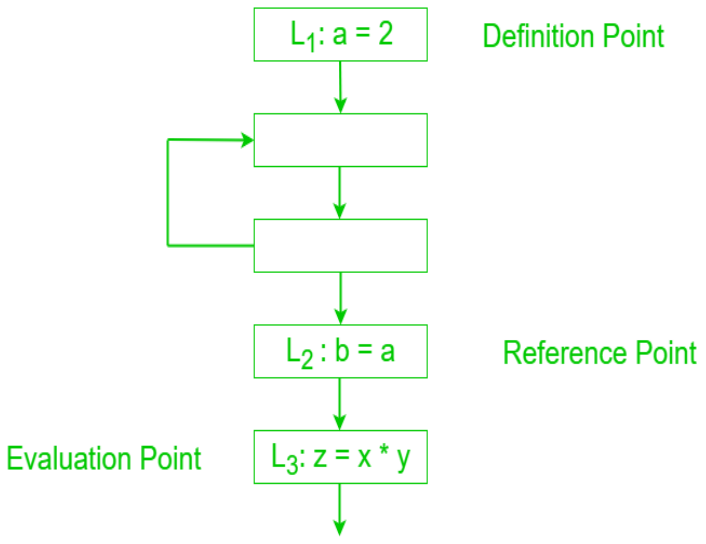
The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like x = x + 1.

## 5.7. Data Flow Analysis:

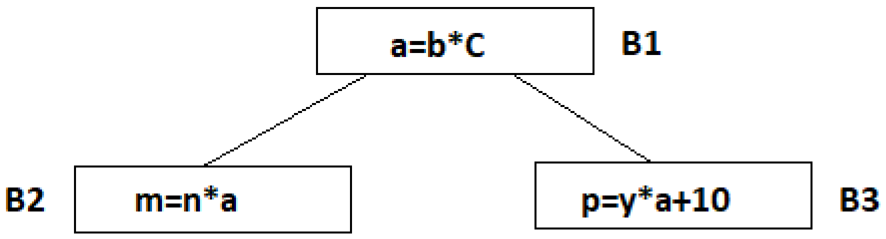
Data flow analysis is a technique of gathering all the information about the program and distributing this information to all the blocks of a flow graph.

It determines the information regarding the definition and use of data in program. This technique is used for optimization.

- **Definition Point:** A point X in a program which contains definition or where definition is specified.
- **Reference Point**: A point X in a program where data item is referred.
- **Evaluation Point:** A point X in a program where an expression is evaluated.

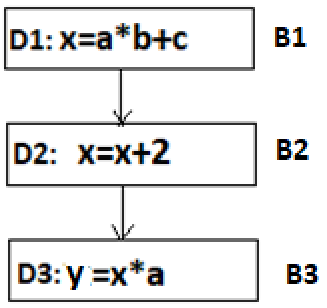


**Available Expression:** An expression is said to be available at program point X if along the path it reaches X. Also an expression is said to be available if none of its operands gets modified before its use.



In above figure expression b*C is available in Blocks B2 and B3. It is used to eliminate common sub expressions.

**Reaching Definition:** A definition D reaches point X, if there is a path from the point immediately following D to X such that D is not killed or not redefined along that path.



D1 is a reaching definition for B2 but not for B3 because it is killed by D2.

**Live Variable**

A variable **v** is live at point **p** if the value of v is used along some path in the flow graph starting at p. Otherwise, the variable is dead.

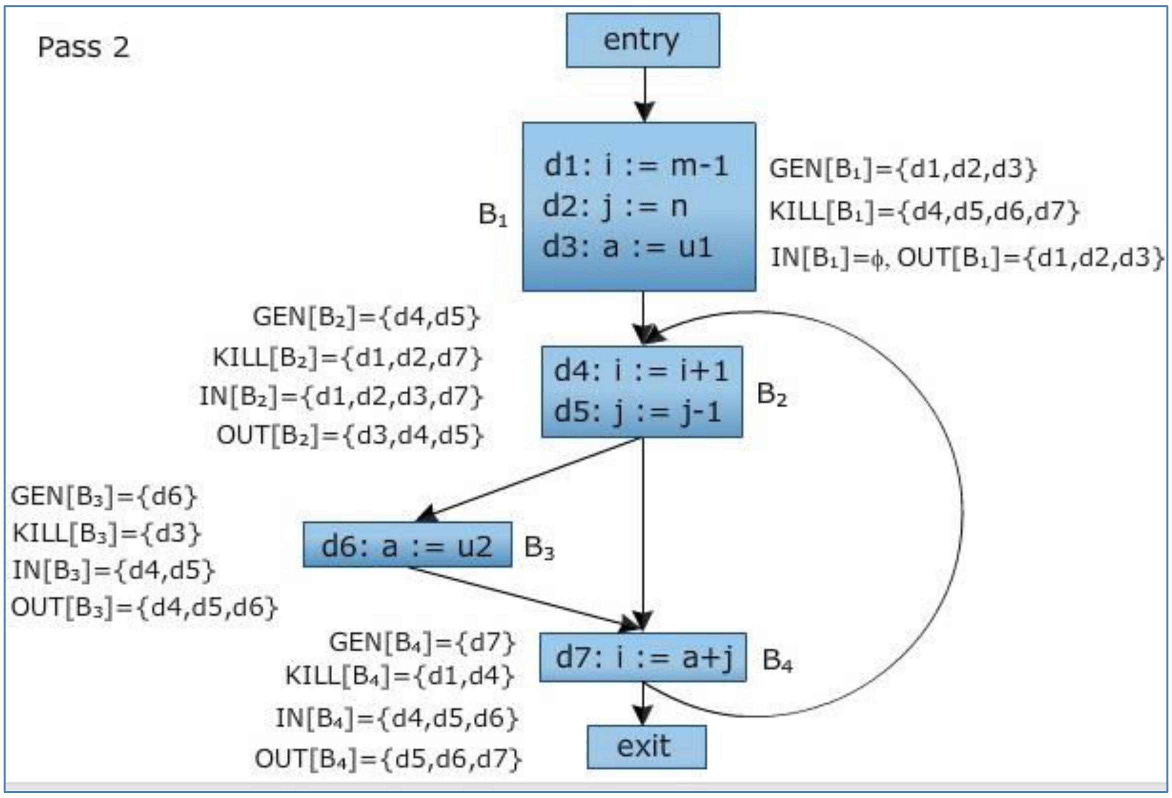
**Data Flow Analysis Equation:**

**OUT[B]= GEN[B] U { IN[B] – KILL[B] }**

**IN[B]=** If some definition reaches B1 entry then IN[B1] is initialised to that set.

**GEN[B]=** set of all definition defined inside B and that are visible after that block.

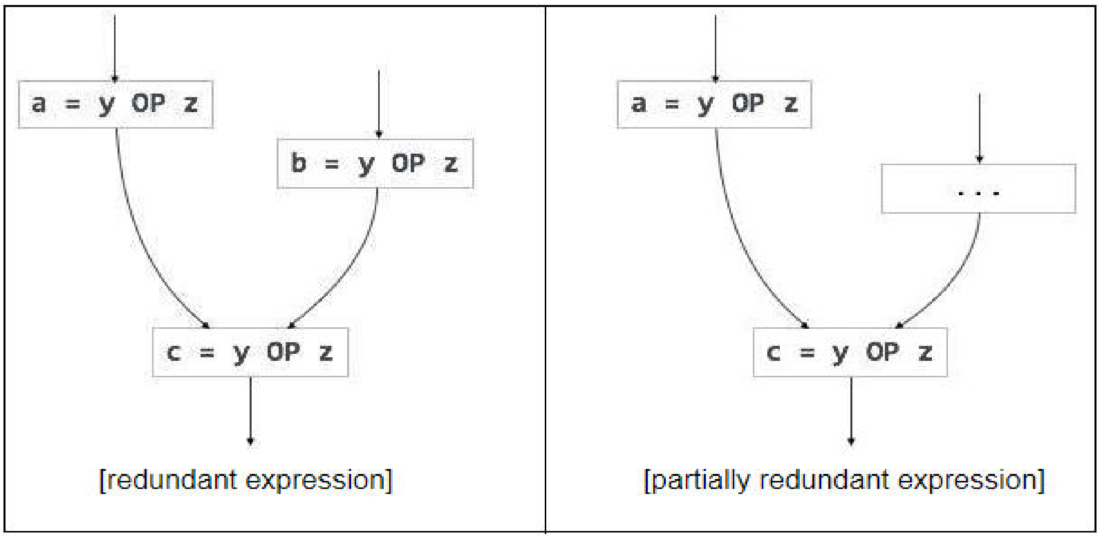
**KILL[B]=**Union of all definitions in all the basic blocks of flow graph that are killed by individual statements in B.



## 5.8. Partial Redundancy Elimination

Partially redundancy elimination performs common sub expression elimination and loop-invariant code motion.

Redundant expressions are computed more than once in parallel path, without any change in operands. Whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique. Another example of a partially redundant code can be:

```
if (condition)
{
  a = y OP z;
}
else
```

```
{
  ...
}
c = y OP z;
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
if (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
else
{
  ...
  tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

**5.8.1. Lazy Code Motion:**

The process of eliminating partial redundancy by delaying the computations as much as possible is called lazy code motion

```
if(condition)
{
// code which does not alter j
i=j=1;
}
else
{
//code which does not alter j
}
k=j+1;
```

This code is partially redundant as the expression j + 1 is computed twice in a condition controlled loop. To optimise this code, we can use partial redundant elimination through loop-invariant code motion and common subexpression elimination to produce the following optimised code:

```
if(condition)
{
// code which does not alter j
n=j=1;
i=n;
}
else
{
//code which does not alter j
n=j+1;
}
k=n;
```

Here, we have removed the redundant assignment and calculation of k = j + 1, instead storing j + 1 inside the temporary variable n and only computed j + 1 once, increasing performance.

## 5.9. Loops in Flow Graph

- Loops are important because programs spend most of their time executing them, and optimizations that improve the performance of loops can have a significant impact. Thus, it is essential that we identify loops and treat them specially. Loops also affect the running time of program analyses.
- Loop analysis is based on the dominators. Dominators are used to determine the loops in a control flow graph.
- **Dominators:** We say node d of a flow graph dominates node n, written **d dom n**, if every path from the entry node of the flow graph to n goes through d. Note that under this definition, every node dominates itself. Consider the flow control graph

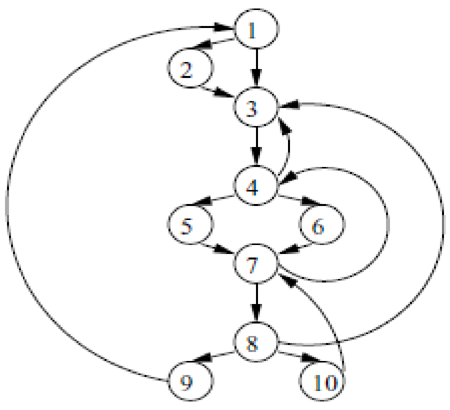


*Fig. Control Flow Graph*

Consider the flow graph in above figure with entry node 1. The entry node dominates every node (this statement is true for every flow graph). Node 2 dominates only itself, since control can reach any other node along a path that begins with 1 -> 3. Node 3 dominates all but 1 and 2. Node 4 dominates all but 1, 2 and 3, since all paths from 1 must begin with l → 2 →3 → 4 or 1 3 4. Nodes 5 and 6 dominate only themselves, since flow of control can skip around either by going through the other. Finally, 7 dominates 7, 8, 9, and 10; 8 dominates 8, 9, and 10; 9 and 10 dominate only themselves.

Dominator information can be represented in a tree called the dominator tree. In this tree, the entry node is the root, and each node d dominates only its descendants.

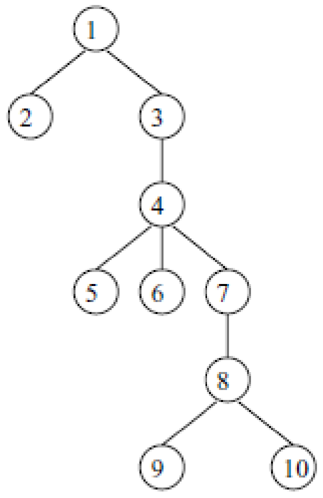


*Fig. Dominator Tree*

The existence of dominator trees follows from a property of dominators: each node n has a unique immediate dominator m that is the last dominator of n on any path from the entry node to n

**Frequently Asked Questions**

1. What is Optimization? Explain various semantic preserving transformations with example.
2. Explain about Loop Optimization with suitable examples.
3. Discuss the importance of Instruction scheduling in Optimization.
4. Explain about various Peephole Optimization Techniques with suitable examples
5. Discuss the importance of Data Flow Analysis with example.